2. Numpy with images

All images are essentially matrices with a variable number of dimensions where each element represents the value of one pixel. The different dimensions and the pixel values can have very different meanings depending on the type of image considered, but the structure is the same.

Python does not allow by default to gracefully handle multi-dimensional data. In particular it is not desgined to handle matrix operations. Numpy was developed to fill in this blank and offers a very similar framework as the one offered by Matlab. It is underlying a large number of packages and has become abolsutely essential to Python scientific programming. In particular it underlies the functions of scikit-image. The latter in turn forms the basis of other software like CellProfiler. It is thus essential to have a good understanding of Numpy to proceed.

Instead of introducing Numpy in an abstract way, we are going here to present it through the lense of image processing in order to focus on the most useful features in the context of this course.

2.1 Exploring an image

Some test images are provided directly in skimage, so let us look at one (we'll deal with the details of image import later). First let us import the necessary packages.

In [1]:
```
import numpy as np
import skimage
import matplotlib.pyplot as plt
plt.gray();  # MZ: nsure it will use gray scale for the plotting
```

In [2]:
```
image = skimage.data.coins()

# submodule skimage.data => provide images
```

In [3]:
```
# MZ: added to have all outputs
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
a=5
a
b=2
b
# => will print 5 and 2 and not only 2
```

Out[3]: 2

2.1.1 Image size

The first thing we can do with the image is simply look at the output:

In [4]:
```
image # MZ: it is a numpy arrray
```

Out[4]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

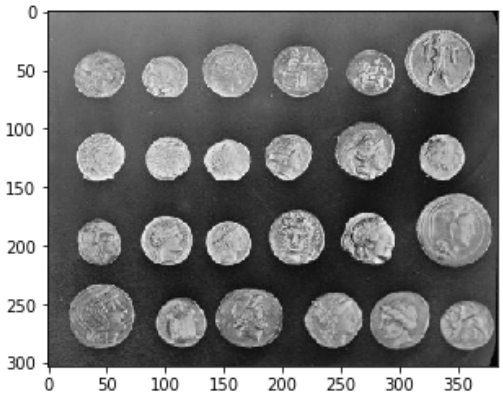
We see that Numpy tells us we have an `array` and we don't have a simple list of pixels, but a *list of lists* representing the fact that we are dealing with a two-dimensional object. Each list represents one row of pixels. Numpy smartly only shows us the first/last rows/columns. We can use the `.shape` method to check the size of the array:

In [5]:
```
image.shape  # MZ: give the dimension
```

Out[5]:
```
(303, 384)
```

This means that we have an image of 303 rows and 384 columns. We can also visualize the image using matplotlib:

In [6]:
```
plt.imshow(image);
```



In [7]:
```
%matplotlib inline
# %matplotlib notebook
# with notebook -> you can zoom, convenient for notebook
# MZ: magic lines for jupyter with %
```

2.1.2 Image type

In [8]:
```
image
```

Out[8]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

In the output above we see that we have one additional piece of information: the array has `dtype = uint8` , which means that the image is of type *unsigned integer 8 bit*. We can also get the type of an array by using:

In [9]:
```
image.dtype  # MZ: dtype is an attribute of "image" (// shape)
```

Out[9]:
```
dtype('uint8')
```

Standard formats we are going to see are 8bit (uint8), 16bit (uint16) and non-integers (usually float64). The type of the image pixels set what values they can take. For example 8bit means values from 0 to $2^8 - 1 = 256 - 1 = 255$. Just like for example in Fiji, one cane change the type of the image. If we know we are going to do operations requiring non-integers we can turn the pixels into floats trough the `.astype()` function.

In [10]:
```
# MZ:
# a bit more careful with types of images !
# if integer or not it really matters !
# numpy different from Python philosophy and dynamic typing
# be careful, e.g. if values > 255 -> can behave weird
```

In [11]:
```
image_float = image.astype(float)
```

Notice the '.':

In [12]:
```
image_float
```

Out[12]:
```
array([[ 47., 123., 133., ...,  14.,   3.,  12.],
       [ 93., 144., 145., ...,  12.,   7.,   7.],
       [126., 147., 143., ...,   2.,  13.,   3.],
       ...,
       [ 81.,  79.,  74., ...,   6.,   4.,   7.],
       [ 88.,  82.,  74., ...,   5.,   7.,   8.],
       [ 91.,  79.,  68., ...,   4.,  10.,   7.]])
```

In [13]:
```
image_float.dtype
```

Out[13]:
```
dtype('float64')
```

The importance of the image type goes slightly against Python's philosophy of dynamics typing (no need to specify a type when creating a variable), but a necessity when handling images. We are going to see now what types of operations we can do with arrays, and the importance of *types* is going to be more obvious.

2.2 Operations on arrays

2.2.1 Arithmetics on arrays

Numpy is written in a smart way such that it is able to handle operations between arrays of different sizes. In the simplest case, one can combine a scalar and an array, for example through an addition:

In [14]:
```
image
```

Out[14]:
```
array([[ 47, 123, 133, ...,  14,   3,  12],
       [ 93, 144, 145, ...,  12,   7,   7],
       [126, 147, 143, ...,   2,  13,   3],
       ...,
       [ 81,  79,  74, ...,   6,   4,   7],
       [ 88,  82,  74, ...,   5,   7,   8],
       [ 91,  79,  68, ...,   4,  10,   7]], dtype=uint8)
```

In [15]:
```
image+10 # add 10 to each element of the array
# MZ:  advantage of using nupy ! will not work with list ! here it works
pixel-wise
```

Out[15]:
```
array([[ 57, 133, 143, ...,  24,  13,  22],
       [103, 154, 155, ...,  22,  17,  17],
       [136, 157, 153, ...,  12,  23,  13],
       ...,
       [ 91,  89,  84, ...,  16,  14,  17],
       [ 98,  92,  84, ...,  15,  17,  18],
       [101,  89,  78, ...,  14,  20,  17]], dtype=uint8)
```

Here Numpy automatically added the scalar 10 to **each** element of the array. Beyond the scalar case, operations between arrays of different sizes are also possible through a mechanism called broadcasting. This is an advanced (and sometimes confusing) features that we won't use in this course but about which you can read for example here (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html).

The only case we are going to consider here is operations between arrays of same size. For example we can multiply the image by itself. We use first the float version of the image:

In [16]:
```
image_sq = image_float*image_float
# MZ:
# does not perform matrix multiplication !, but multiply each pixel with
each pixel at the same position
# (will not perform like in linear algebra) (will have to use other nump
y functions)
```

In [17]:
```
image_sq
```

Out[17]:
```
array([[2.2090e+03, 1.5129e+04, 1.7689e+04, ..., 1.9600e+02, 9.0000e+00,
        1.4400e+02],
       [8.6490e+03, 2.0736e+04, 2.1025e+04, ..., 1.4400e+02, 4.9000e+01,
        4.9000e+01],
       [1.5876e+04, 2.1609e+04, 2.0449e+04, ..., 4.0000e+00, 1.6900e+02,
        9.0000e+00],
       ...,
       [6.5610e+03, 6.2410e+03, 5.4760e+03, ..., 3.6000e+01, 1.6000e+01,
        4.9000e+01],
       [7.7440e+03, 6.7240e+03, 5.4760e+03, ..., 2.5000e+01, 4.9000e+01,
        6.4000e+01],
       [8.2810e+03, 6.2410e+03, 4.6240e+03, ..., 1.6000e+01, 1.0000e+02,
        4.9000e+01]])
```

In [18]:
```
image_float
```

Out[18]:
```
array([[ 47., 123., 133., ...,  14.,   3.,  12.],
       [ 93., 144., 145., ...,  12.,   7.,   7.],
       [126., 147., 143., ...,   2.,  13.,   3.],
       ...,
       [ 81.,  79.,  74., ...,   6.,   4.,   7.],
       [ 88.,  82.,  74., ...,   5.,   7.,   8.],
       [ 91.,  79.,  68., ...,   4.,  10.,   7.]])
```

Looking at the first row we see $47^2 = 2209$ and $123^2 = 15129$ etc. which means that the multiplication operation has happened **pixel-wise**. Note that this is **NOT** a classical matrix multiplication. We can also see that the output has the same size as the original arrays:

In [19]:
```
image_sq.shape
```

Out[19]:
```
(303, 384)
```

In [20]:
```
image_float.shape
```

Out[20]:
```
(303, 384)
```

Let's see now what happens when we square the original 8bit image:

In [21]:
```
image*image
```

Out[21]:
```
array([[161,  25,  25, ..., 196,   9, 144],
       [201,   0,  33, ..., 144,  49,  49],
       [  4, 105, 225, ...,   4, 169,   9],
       ...,
       [161,  97, 100, ...,  36,  16,  49],
       [ 64,  68, 100, ...,  25,  49,  64],
       [ 89,  97,  16, ...,  16, 100,  49]], dtype=uint8)
```

We see that we don't get at all the expected result. Since we multiplied two 8bit images, Numpy assumes we want an 8bit output. And therefore the values are bound between 0-255. For example the first value is just the remainder of the modulo 256:

In [22]:
```
# MZ:
# what is above 255 get reassigned to a 0-255 value
# as numpy assumed that we have 8bit int !!!

# if you want > 255 values -> first make the matrix as float
```

In [23]:
```
2209%256
```

Out[23]:
```
161
```

The same thing happens e.g. if we add an integer scaler to the matrix:

In [24]:
```
print(image+230)
```

```
[[ 21  97 107 ... 244 233 242]
 [ 67 118 119 ... 242 237 237]
 [100 121 117 ... 232 243 233]
 ...
 [ 55  53  48 ... 236 234 237]
 [ 62  56  48 ... 235 237 238]
 [ 65  53  42 ... 234 240 237]]
```

Clearly something went wrong as we get values that are smaller than 230. Again any value "over-flowing" above 255 goes back to 0.

This problem can be alleviated in different ways. For example we can combine a integer array with a float scaler and Numpy will automatically give a result using the "most complex" type:

In [25]:
```
image_plus_float = image+230.0
```

In [26]:
```
print(image_plus_float)  # MZ: e.g. has removed 256: 277-256 = 21
```

```
[[277. 353. 363. ... 244. 233. 242.]
 [323. 374. 375. ... 242. 237. 237.]
 [356. 377. 373. ... 232. 243. 233.]
 ...
 [311. 309. 304. ... 236. 234. 237.]
 [318. 312. 304. ... 235. 237. 238.]
 [321. 309. 298. ... 234. 240. 237.]]
```

To be on the safe side we can also explicitely change the type when we know we might run into this kind of trouble. This can be done via the `.astype()` method:

In [27]:
```
# MZ:
# combine integer with float -> Python logic, use the most complex type
# will convert int to float and the output will be float
```

In [28]:
```
image_float = image.astype(float)
```

In [29]:
```
image_float.dtype
```

Out[29]:
```
dtype('float64')
```

Again, if we combine floats and integers the output is going to be a float:

In [30]:
```
image_float+230
```

Out[30]:
```
array([[277., 353., 363., ..., 244., 233., 242.],
       [323., 374., 375., ..., 242., 237., 237.],
       [356., 377., 373., ..., 232., 243., 233.],
       ...,
       [311., 309., 304., ..., 236., 234., 237.],
       [318., 312., 304., ..., 235., 237., 238.],
       [321., 309., 298., ..., 234., 240., 237.]])
```

2.2.2 Logical operations

A set of important operations when processing images are logical (or boolean) operations that allow to create masks for features to segment. Those have a very simple syntax in Numpy. For example, let's compare pixel intensities to some value *a*:

In [31]:
```
threshold = 100
```

In [32]:
```
image > threshold
```

Out[32]:
```
array([[False,  True,  True, ..., False, False, False],
       [False,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

We see that the result is again a pixel-wise comparison with a, generating in the end a boolean or logical matrix. We can directly assign this logical matrix to a variable and verify its shape and type and plot it:

In [33]:
```
image_threshold = image > threshold
```

In [34]:
```
image_threshold.shape
```

Out[34]:
```
(303, 384)
```

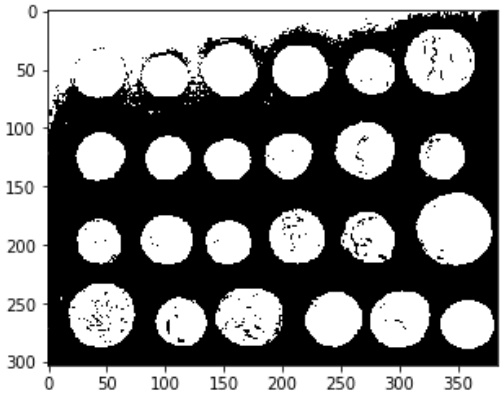
In [35]:
```
image_threshold.dtype
```

Out[35]:
```
dtype('bool')
```

In [36]:
```
image_threshold
```

Out[36]:
```
array([[False,  True,  True, ..., False, False, False],
       [False,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       ...,
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False],
       [False, False, False, ..., False, False, False]])
```

In [37]:
```
plt.imshow(image_threshold);
```



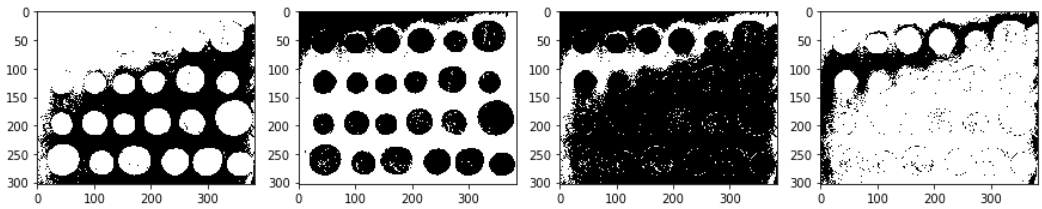
Of course other logical operator can be used (<, >, ==, !=) and the resulting boolean matrices combined:

In [38]:
```
threshold1 = 70
threshold2 = 100
image_threshold1 = image > threshold1
image_threshold2 = image < threshold2
```

In [39]:
```
# MZ
# logical: often use of masks
# e.g. you have a mask for dog and a mask for houses -> apply the masks
to the images using logicals
```

In [40]:
```
# MZ: here we deal with logical matrices
image_AND = image_threshold1 & image_threshold2  # MZ: True in the 2 mat
rices
image_XOR = image_threshold1 ^ image_threshold2  # MZ: what is True in 1
matrix but not in the other one
```

In [41]:
```
# MZ: multiple panels on matplot
plt.figure(figsize=(15,15)) # set the figure sizes
plt.subplot(1,4,1) # how many 1 row, 4 columns, and what is the 1st elem
ent
plt.imshow(image_threshold1)
plt.subplot(1,4,2) # in the subplot where 1 row and 4 columns, what shou
ld be the 2nd element
plt.imshow(image_threshold2)
plt.subplot(1,4,3)
plt.imshow(image_AND)
plt.subplot(1,4,4)
plt.imshow(image_XOR);
```



2.3 Numpy functions

To broadly summarize, one can say that Numpy offers three types of operations: 1. Creation of various types of arrays, 2. Pixel-wise modifications of arrays, 3. Operations changing array dimensions, 4. Combinations of arrays.

2.3.1 Array creation

Often we are going to create new arrays that later transform them. Functions creating arrays usually take arguments spcifying both the content of the array and its dimensions.

Some of the most useful functions create 1D arrays of ordered values. For example to create a sequence of numbers separated by a given step size:

In [42]:
```
np.arange(0,20,2)  # MZ: from where to where in step of what
```

Out[42]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

Or to create an array with a given number of equidistant values:

In [43]:
```
np.linspace(0,20,5)
```

Out[43]:
```
array([ 0.,  5., 10., 15., 20.])
```

In higher dimensions, the simplest example is the creation of arrays full of ones or zeros. In that case one only has to specify the dimensions. For example to create a 3x5 array of zeros:

In [44]:
```
np.zeros((3,5))
```

Out[44]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

Same for an array filled with ones:

In [45]:
```
np.ones((3,5))
```

Out[45]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

Until now we have only created one-dimensional lists of 2D arrays. However Numpy is designed to work with arrays of arbitrary dimensions. For example we can easily create a three-dimensional "ones-array" of dimension 5x8x4:

In [46]:
```
array3D = np.ones((2,6,5))
```

In [47]:
```
array3D
```

Out[47]:
```
array([[[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]],

       [[1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1.]]])
```

In [48]:
```
array3D.shape
# MZ: you should decide which dimension is the channel/volume (usually t
he 1st or the last)

# MZ: numpy functions can easily deal with any dimension
# (e.g. it is easy to convert code written for 2D to code for 3D object
s)
```

Out[48]:
```
(2, 6, 5)
```

And all operations that we have seen until now and the following ones apply to such high-dimensional arrays exactly in the same way as before:

In [49]:
```
array3D*5
```

Out[49]:
```
array([[[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]],

       [[5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.],
        [5., 5., 5., 5., 5.]]])
```

We can also create more complex arrays. For example an array filled with numbers drawn from a normal distribution:

In [50]:
```
np.random.standard_normal((3,5))
```

Out[50]:
```
array([[ 0.51920188, -1.74490051,  0.19059971, -1.22782172, -0.08362917],
       [-1.91288875, -1.46339209, -0.29266003,  1.58959264,  1.39652976],
       [-2.23327794,  0.4977774 , -0.04227832,  0.97826304, -0.9933275
6]])
```

As mentioned before, some array-creating functions take additional arguments. For example we can draw samples from a gaussian distribution whose mean and variance we can specify.

In [51]:
```
np.random.normal(10, 2, (5,2))
# MZ: NB "Tab" for auto-completion; "Shift+Tab" to get the help for the
function
```

Out[51]:
```
array([[11.59504334, 10.84820206],
       [11.21592976,  9.46107067],
       [ 8.06999708, 10.02220069],
       [10.15008664, 11.81826128],
       [ 7.92993365, 11.43523018]])
```

2.3.2 Pixel-wise operations

Numpy has a large trove of functions to do all common mathematical operations matrix-wise. For example you can take the cosine of a matrix:

In [52]:
```
angles = np.random.random_sample(5)
angles
```

Out[52]:
```
array([0.94436116, 0.77710703, 0.8668537 , 0.68759525, 0.25572394])
```

In [53]:
```
np.cos(angles)
```

Out[53]:
```
array([0.58626054, 0.71294513, 0.64722816, 0.7727745 , 0.96748043])
```

Or to calculate exponential values:

In [54]:
```
np.exp(angles)
```

Out[54]:
```
array([2.57117028, 2.17517045, 2.37941273, 1.98892691, 1.29139618])
```

And many many more.

2.2.3 Operations changing dimensions

Some functions are accessible in the form of method, i.e. they are called using the dot notation. For example to find the maximum in an array:

In [55]:
```
angles.max() # MZ: return the max value inside the array
```

Out[55]:
```
0.9443611558667749
```

Alternatively there's also a maximum function:

In [56]:
```
np.max(angles)  # MZ: same as above but calling directly as a function
```

Out[56]: 0.9443611558667749

The `max` function like many others (min, mean, median etc.) can also be applied to a given axis. Let's imagine we have a 3D image (multiple planes) of 10x10x4 pixels:

In [ ]:
```
volume = np.random.random((10,10,4))
#volume
```

If we want to do a maximum projection along the third axis, we can specify:

In [58]:
```
projection = np.max(volume, axis = 2)
# MZ: specify an axis
# 0 1 2
# maximum along the 3 -> axis = 2
# creates a projection
```

In [59]:
```
projection.shape
```

Out[59]: (10, 10)

In [60]:
```
projection2 = np.max(volume, axis = 0)
projection2.shape
```

Out[60]: (10, 4)

In [61]:
```
projection3 = np.max(volume, axis = 1)
projection3.shape
```

Out[61]: (10, 4)

We see that we have indeed a new array with one dimension less because of the projection.

2.3.4 Combination of arrays

Finally arrays can be combined in multiple ways. For example if we want to assemble to images with the same size into a stack, we can use the stack function:

In [62]:
```
image1 = np.ones((4,4))
image2 = np.zeros((4,4))

stack = np.stack([image1, image2],axis = 2)
```

In [63]:
```
stack.shape
```

Out[63]: (4, 4, 2)

2.3 Slicing and indexing

Just like broadcasting, the selection of parts of arrays by slicing or indexing can become very sophisticated. We present here only the very basics to avoid confusion. There are often multiple ways to do slicing/indexing and we favor here easier to understant but sometimes less efficient solutions.

To simplify the visualisation, we use here a natural image included in the skimage package.

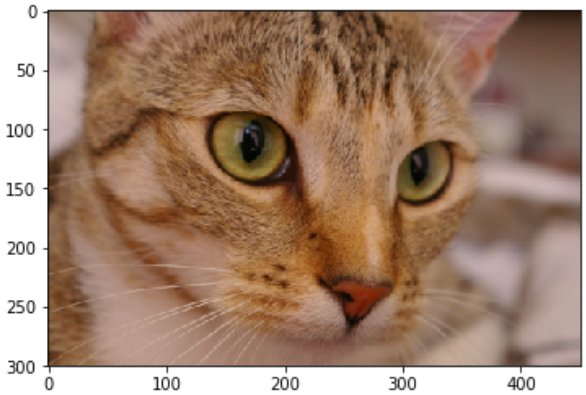
In [64]:
```
image = skimage.data.chelsea()
```

In [65]:
```
image.shape # MZ: 300x451 pixels and 3 planes: RGB
```

Out[65]: (300, 451, 3)

We see that the image has three dimensions, probably it's a stack of three images of size 300x400. Let us try to have a look at this image hoping that dimensions are handled gracefully:

In [66]:
```
plt.imshow(image); # MZ: if pass an image with 3 planes as last dim -> i
mplicitly assumes it is an RGB image
```



So we have an image of a cat with dimensions 300x400. The image being in natural colors, the three dimensions probably indicate an RGB (red, green, blue) format, and the plotting function just knows what to do in that case.

2.3.1 Array slicing

Let us now just look at one of the three planes composing the image. To do that, we are going the select a portion of the image array by slicing it. One can give:

- a single index e.g. 0 for the first element
- a range e.g. 0:10 for the first 10 elements
- take all elements using a semi-column :

What portion is selected has to be specified for each dimensions of an array. In our particular case, we want to select all rows, all columns and a single plane of the image:

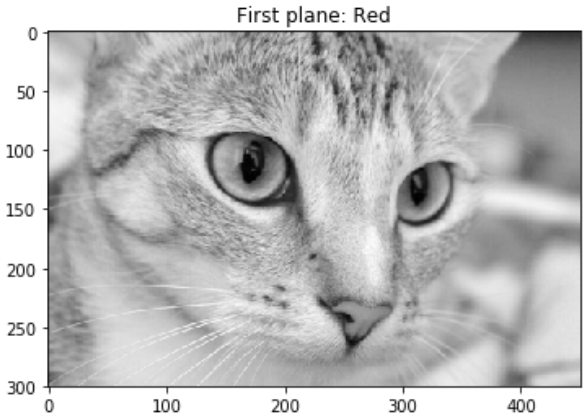
In [67]:
```
image.shape
```

Out[67]: (300, 451, 3)

In [68]:
```
image[:,:,1].shape # MZ: select only the 2nd plane
```

Out[68]: (300, 451)

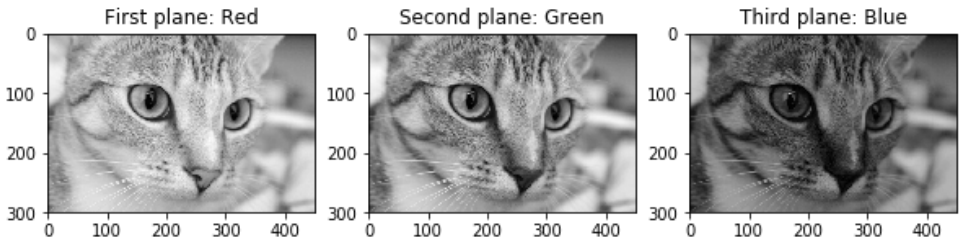
In [69]:
```
plt.imshow(image[:,:,0],cmap='gray')
# MZ: cmap argument -> here redondant with plt.gray();
# different colormaps provided by matplotlib (map pixel-values to color
s)
plt.title('First plane: Red');
```



We see now the red layer of the image. We can do the same for the others by specifying planes 0, 1, and 2:

In [70]:
```
plt.figure(figsize=(10,10))
plt.subplot(1,3,1)
plt.imshow(image[:,:,0],cmap='gray')
plt.title('First plane: Red')
plt.subplot(1,3,2)
plt.imshow(image[:,:,1],cmap='gray')
plt.title('Second plane: Green')
plt.subplot(1,3,3)
plt.imshow(image[:,:,2],cmap='gray')
plt.title('Third plane: Blue');


# MZ:
# no physical meaning to the colormaps, you can put what ever you want a
s colors
# is only the rendering of the pixel values
```



Logically intensities are high for the red channel and low for the blue channel as the image has red/brown patterns. We can confirm that by measuring the mean of each plane. To do that we use the same function as above but apply it to a singel sliced plane:

In [71]:
```
image0 = image[:,:,0] # MZ: retain only 1st dim
```

In [72]:
```
np.mean(image0) # MZ: mean of all pixels
```

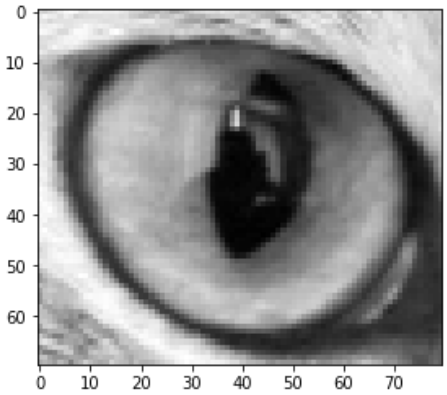
Out[72]:
```
147.67308943089432
```

and for all planes using a comprehension list:

```
In [73]: [np.mean(image[:,:,i]) for i in range(3)]  # MZ: calculat the mean of ev
         ery plane
Out[73]: [147.67308943089432, 111.44447893569844, 86.79785661492978]
```

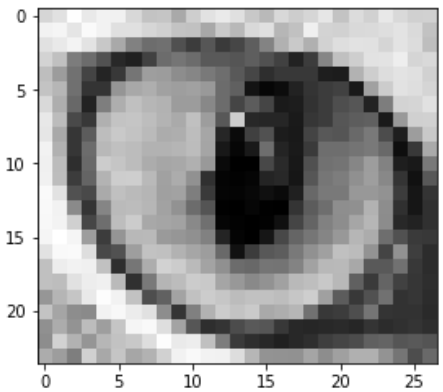
To look at some more details let us focus on a smaller portion of the image e.g. one of the cat's eyes. For that we are going to take a slice of the red image and store it in a new variable and display the selection. We consider pixel rows from 80 to 150 and columns from 130 to 210 of the first plane (0).

```
In [74]: image_red = image[80:150,130:210,0]
         plt.imshow(image_red,cmap='gray');
```



There are different ways to select parts of an array. For example one can select every n'th element by giving a step size. In the case of an image, this subsamples the data:

```
In [75]: image_subsample = image[80:150:3,130:210:3,0]
         plt.imshow(image_subsample,cmap='gray');
```



2.3.2 Array indexing

In addition to slicing an array, we can also select specific values out of it. There are many (https://docs.scipy.org/doc/numpy-1.13.0/reference/arrays.indexing.html) different ways to achieve that, but we focus here on two main ones.

First, one might have a list of pixel positions and one wishes to get the values of those pixels. By passing two lists of the same size containing the rows and columns positions of those pixels, one can recover them:

In [76]:
```
row_position = [0,1,2,3]
col_position = [0,1,0,1]

print(image_red[0:5,0:5])
# MZ: pass the 2 lists -> assumes that you mean the pixels you want

image_red[row_position,col_position]
# MZ: output is just a list of pixels, not in 3 dim anymore ! output is
1D

# MZ => you can extract either with 3-dot notation or by passing a list
```

```
[[166 162 169 174 185]
 [183 192 185 183 173]
 [179 178 168 175 176]
 [187 184 187 189 185]
 [195 192 187 181 169]]
```

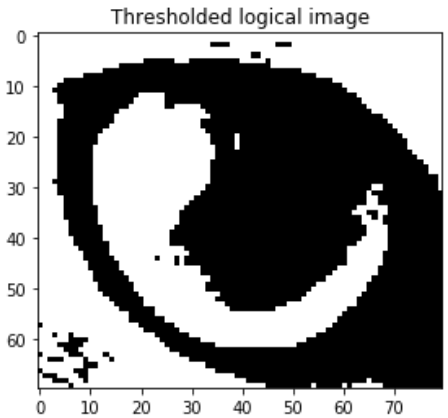
Out[76]:
```
array([166, 192, 179, 184], dtype=uint8)
```

Alternatively, one can pass a logical array of the same dimensions as the original array, and only the True pixels are selected. For example, let us create a logical array by picking values above a threshold:

In [77]:
```
threshold_image = image_red>120
```

Let's visualize it. Matplotlib handles logical arrays simply as a binary image:

In [78]:
```
plt.imshow(threshold_image)
plt.title('Thresholded logical image');
```



We can recover the value of all the "white" (True) pixels in the original image by **indexing one array with the other**:

In [79]:
```
selected_pixels = image_red[threshold_image]
# MZ:
# create a mask with logical array
# pass another image, of the same size, should be a boolean array and
# instead of passing explicit lists of rows/columns -> direct pass an ar
ray
# output is again a list
# useful e.g. for segmentation (create a mask where you have the cells o
nly to extract
# from other panes where you have light emission and average the light e
mission)
print(selected_pixels)
```

```
[166 162 169 ... 148 137 132]
```

And now ask how many pixels are above threshold and what their average value is.

In [80]:
```
len(selected_pixels)
```

Out[80]: 2585

In [81]:
```
np.mean(selected_pixels)
```

Out[81]: 153.59381044487426

In [82]:
```
threshold_image # MZ: mask is a boolean array 2D
```

Out[82]:
```
array([[ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       [ True,  True,  True, ...,  True,  True,  True],
       ...,
       [ True, False, False, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False],
       [ True,  True,  True, ..., False, False, False]])
```

In [83]:
```
np.argwhere(threshold_image)
# MZ: 2 dim arrays -> gives where are the True values in x,y coordinates
```

Out[83]:
```
array([[ 0,  0],
       [ 0,  1],
       [ 0,  2],
       ...,
       [69, 65],
       [69, 66],
       [69, 67]])
```

In [ ]:
```
# MZ: to have all attributes and functions associated with an object
#dir(threshold_image)
```

In [ ]:
```
# MZ: same works for packages
#dir(np)
```

We now know that there are 2585 pixels above the threshold and that their mean is 153.6

In [86]:
```
# to plot with transprency: (e.g. to plot 1 fig on the top of another)
# imshow(alpha=0.5)
```